Interprocedural Analysis for Loop Scheduling and Data Allocation

Trung N. Nguyen [a], Zhiyuan Li [b] [1]

[a] *IBM, Rochester, Minnesota 55906, USA*

[b] *Department of Computer Science, Purdue University, W. Lafayette, IN 47907, USA*

Abstract

In order to reduce remote memory accesses on CC-NUMA multiprocessors, we present an interprocedural analysis to support static loop scheduling and data allocation. Given a parallelized program, the compiler constructs graphs which represent globally and interprocedurally the remote reference penalties associated with different choices for loop scheduling and data allocation. After deriving an optimal solution according to those graphs, the compiler generates data allocation directives and schedules DOALL loops. Experiments indicate that the proposed compiler scheme is efficient and simulation results show good performance of the parallel code.

---

1 Introduction

Executing independent loop iterations on multiple processors is an important approach to improving program execution speed. A loop which contains sufficient parallelism among its iterations can be parallelized either manually by a programmer or, in many cases, automatically by a compiler. Once a loop is transformed into a parallel form, its iterations are assigned to different processors for execution, using a certain *loop scheduling* method. Static scheduling methods perform such assignment at compile time, while dynamic scheduling methods leave the assignment decision to program execution time.

[1] Corresponding author. E-mail li@cs.purdue.edu.

The Silicon Graphics Challenge multiprocessor, for example, provides two static scheduling methods, *simple* and *interleave*, and two dynamic scheduling methods, *gss* and *dynamic* [13]. The *simple* scheduling method, also known as *block* scheduling, divides the loop iterations by the number of processors and then assigns each chunk of consecutive iterations to one processor. The *interleave* scheduling method divides the iterations into chunks of the size specified by the CHUNK option, and execution of those chunks is statically interleaved among the processors. With *dynamic* scheduling, the iterations are also divided into CHUNK-sized chunks. As each processor finishes a chunk, however, it enters a critical section to grab the next available chunk. With *gss* scheduling [25] , the chunk size is varied, depending on the number of iterations remaining.

The speedup of parallel loop execution on shared memory machines is determined by program parallelism, data locality, scheduling and synchronization overhead, and load balance. Many parallel loops in well-known benchmarking programs are found to have little variance in the workload of different iterations and such loops can be recognized by the compiler [9, 22]. Static loop scheduling is most suitable for such programs, not only because it reduces the scheduling overhead, but also because it can enhance data locality. The compiler can assign loop iterations which frequently access the same memory locations to the same processor [4, 15, 27, 19, 12, 18, 28]. Moreover, on cache-coherent non-uniform memory-access (CC-NUMA) multiprocessors, data frequently accessed in a particular loop iteration can be statically allocated to the processor that executes the iteration. In the event of cache misses, data in the local memory will be faster to access than those in remote memories. Data allocation and loop scheduling often need to be considered simultaneously in order to achieve the best results.

This paper presents a new compiler technique which allocates data and schedules DOALL parallel loops simultaneously for CC-NUMA machines. Unlike several previous works which consider one routine at a time, this technique examines DOALL loops in all routines in the program and coordinate their scheduling and data allocation altogether. Another new aspect in our approach is that we do not sacrifice DOALL loop parallelism in exchange for reduced remote memory references as some previous works do. Based on the parallelism available on the given machine, the compiler selects certain DOALL loops to execute in parallel, even if remote references may result.

The DOALL loop selection, however, minimizes remote references among all potential choices.

After DOALL loops are selected, the compiler determines how to *align* the iterations in different DOALL loops and how to align data with loop iterations. Based on the analysis result, the compiler performs loop scheduling and generates data allocation directives. We perform experiments within an advanced interprocedural parallelizing compiler, Panorama [10], and conduct execution-driven simulations using a detailed CC-NUMA simulator.

In the rest of this paper, we first review related work (Section 2) and give an overview of Panorama (Section 3). We then present a graph-based compiler analysis to determine loop iteration assignment and data allocation (Section 4). In Section 5, we discuss the issue of parameter mapping during the interprocedural analysis. Both Section 4 and Section 5 contain static program statistics collected in our implementation. In Section 6, we present simulation results and summarize this work.

2 Related Work

This work targets shared-memory multiprocessors, which is different from previous works by J. Li and M. Chen [20] and by U. Kremer et al. [16] which target distributed memory multicomputers. W. Li and K. Pingali [21] consider loop transformations to enhance data locality, whereas we consider data allocation in conjunction with loop scheduling. Agarwal et al. [1] consider a single loop nest and we considers the whole program.

Our work shares several aspects with the works done by the Stanford SUIF group: We both target the CC-NUMA architecture and both schedule multiple loop nests simultaneously. However, the work by A. Lim and M. Lam [23] addresses a different issue from ours. Their objective is to maximize parallelism and minimize barrier synchronization, whereas ours is to minimize remote memory references subject to the condition that the DOALL loop parallelism available both in the program and on the given machine will not be sacrificed. Our work is closer to the work done by J. Anderson and M. Lam [4] in that their objective is also to reduce remote memory references. The main differences between their work and ours are the following:

- *The main approach* – Their method primarily looks for solutions to a large matrix which represents a desired combination of data allocation and loop scheduling which is communication free. Where such solutions do not exist, they execute some of the loops as *DOACROSS loops*, while serializing some DOALL loops. (Details can be found in [4, 2]). Our method, on the other hand, primarily reduces decision-graphs, which leads to minimal remote references. We execute DOALL loops in parallel but not DOACROSS loops, because the former requires only barrier synchronization, while the latter requires point-to-point data synchronization. Instead of serializing some DOALL loops, we keep as many DOALL loops as the system supports, even if remote references may result.

- *The cost model* – Their cost model is primarily based on the cost of reorganizing data. Ours is primarily based on the counts of remote references, without data reorganization.

- *The scope* – Their work applies to *fully-permutable* loop nests only, and ours applies to arbitrary loop nests, even those nests containing routine calls. We discuss, in this paper, the issue of parameter mapping at the call sites when the compiler coordinates loop scheduling and data allocation across routines. This issue has not been addressed in previous works. A recent paper by J. Anderson, S. Amarasinghe and M. Lam, however, mentions that they have implemented an interprocedural version of their algorithm in SUIF [3]. Anderson and Lam's paper, on the other hand, considers more general array addressing functions than our work. They also consider multi-dimensional processor topology, while we consider one dimension only.

3 The Panorama parallelizing compiler

Panorama is a source-to-source parallelizing Fortran compiler [10] developed when the authors were at the University of Minnesota. It uses a *hierarchical supergraph* (HSG), similar to the HSCG of the the PIPS project [14], to represent program control flow both intraprocedurally and interprocedurally. The HSG provides sufficient interprocedural information to the compiler such that static scheduling and data allocation can be coordinated across procedure boundaries.

The HSG is a composition of the control flow subgraphs of all routines in a program. Each call statement is represented by a *call compound node*, while each DO loop is represented by a *loop compound node*. The resulting graph contains three kinds of nodes – basic block nodes, loop compound nodes and call compound nodes. Each compound node has an attached flow subgraph describing the control flow within the called routine or the DO loop. A HSG for a sample program is shown in Figure 1. 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, thus avoiding compiler inefficiency introduced by *in-lining*. Each node in the HSG is annotated by important information regarding the array reference patterns in this node. This proves not only important to our array dataflow analysis but also to the alignment of DOALL loop iterations. As far as we know, the use of an interprocedural control flow graph and its associated array reference information for static loop scheduling and data allocation is unique to this work.

Panorama performs efficient interprocedural array data flow analysis to enable array privatization and loop parallelization [10]. Loop compound nodes are marked as DOALL loops if the represented loops do not contain loop-carried dependences after program transformations. A set of symbolic evaluation utility routines are implemented to support array data flow analysis. The work presented in this paper utilizes one of these routines to compare the value ranges of arithmetic expressions. Each of such expressions is normalized as a sum of products. Our experience suggests that addition, subtraction, and multiplication are sufficient for most cases. For example, to compare which of two symbolic expressions is greater, we simply subtract one from the other and then check to see if the result is greater than zero.

4 Task Alignment and Data Allocation

On shared memory multiprocessors which have private caches on each processor, an important way to enhance the cache hit ratio in DOALL loops is to perform *task alignment* during loop scheduling, i.e., to assign those iterations in different DOALL loops to the same processor if they frequently access many common memory locations. On CC-NUMA multiprocessors, it is also important to allocate data items to the processor which accesses

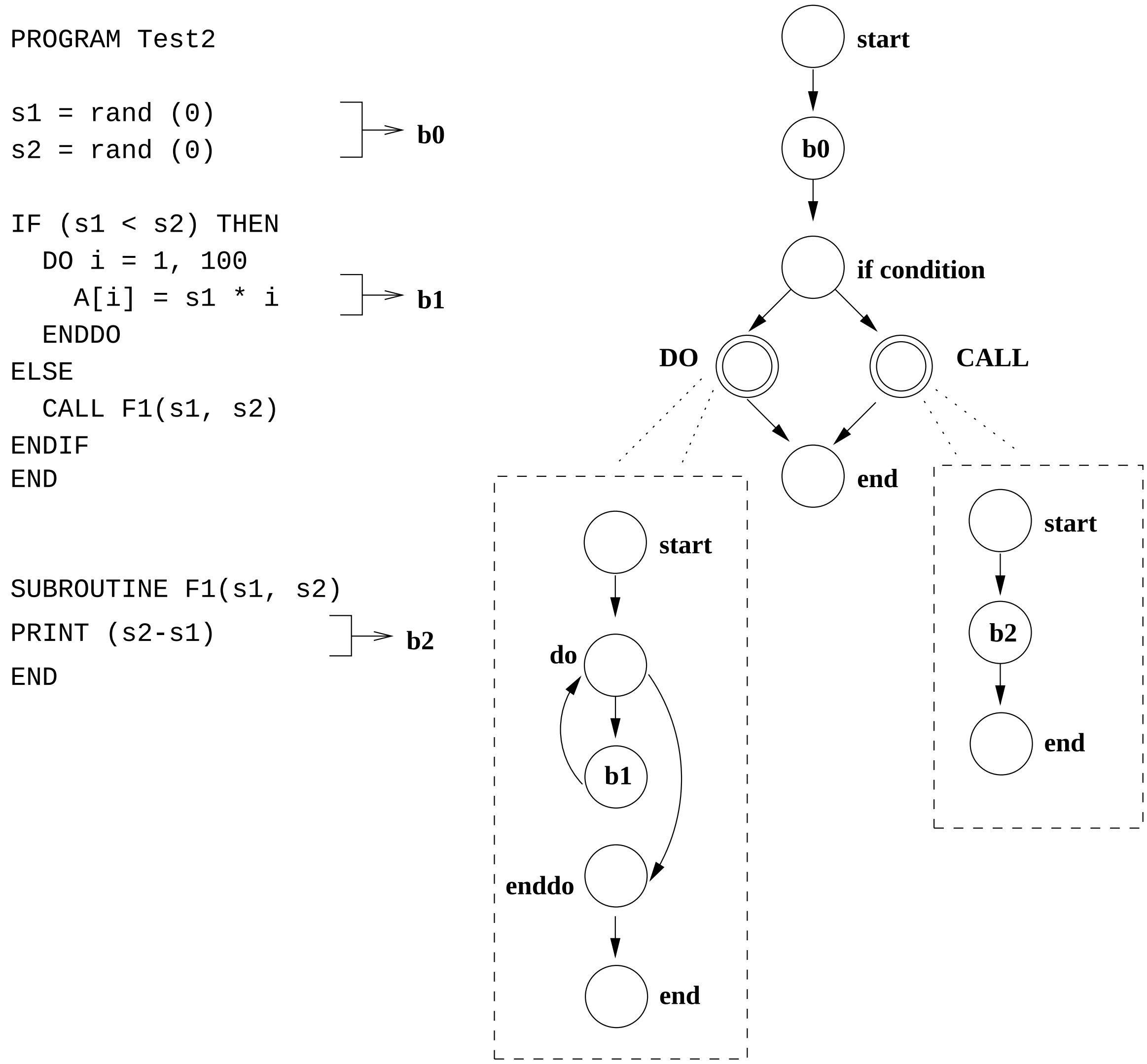


Figure 1: An example of a hierarchical supergraph. Single circles represent basic blocks while double circles represent compound nodes.

them most frequently. In the event of a cache miss, the processor can then find the data in its local memory unit. Take the following example.

```
DOALL 1 i = 1, n
   a(i) = b(i) + c(i)
ENDDO
...
DOALL 2 i = 1, n
   d(i) = a(i) * b(i)
ENDDO
```

For the program listed above, the compiler can schedule iteration i in each of the two DOALL loops to the same processor where $a(i)$ resides. Unfortunately, it is quite often that a data item will be accessed by different processors no matter how the loop iterations are scheduled to different processors. One desires to find a solution which minimizes the number of references to remote memories, without sacrificing the parallelism available both in the program and in the system. In this section, we present a solution that is based on interprocedural analysis and decision-graph reduction.

We assume an abundance of *virtual processors* (VP's) and assign the DOALL 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 CC-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.

For the sake of discussions, we assume in this paper that all DOALL loops are normalized to have lower loop limits of one and index steps of one. Similarly, we assume the lower bound of each array dimension to be one. For each array reference, we assume the index expression in each array dimension to be in the form of $i+e$, $-i+e$, or e, where i is a loop index and e is a symbolic expression which is i-loop invariant. Array references which do not fit these assumed patterns are currently excluded from consideration, which may potentially degrade the quality of the assignment decisions. The graph construction algorithms presented in this work can be extended to accommodate a wider class of array references if such extension is needed in practice.

As Anderson and Lam pointed out [4], the alignment problem can be broken down into two optimization phases:

1. Phase I: Optimizing the *orientation* of the data allocation and the assignment of the parallel iterations. Suppose the VP's are numbered from 1 to P. 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 the decreasing order. During this phase, we also select one loop among a nest of multiple DOALL loops to execute in parallel while sequentializing the others. (This is because most commercial CC-NUMA multiprocessor systems currently support singly-nested DOALL loops only.) In addition, for each multi-dimensional array, we choose one dimension as the one to be distributed among the VP's. The processes of selecting loop orientation, array dimension and orientation, and selecting which DOALL loop in a multiply-nested DOALL to keep are all performed simultaneously in this phase.

2. Phase II: Optimizing the *displacement* of the chosen orientations. Displacement means how far, if at all, we should shift the data assignment or the iteration assignment to the left or to the right. For a displacement of c in the increasing orientation, a loop iteration i is assigned to the VP numbered $i + c$. For the same displacement, c, but in the decreasing orientation, this same iteration would be assigned to the VP numbered $P + 1 - i + c$. If a displacement is small, then its effect is not as important as choosing a good orientation because its effect usually diminishes due to large data cache lines in practice.

4.1 Constructing the connection graph

Before starting the two optimizing phases, we create a *connection graph* [4] to allow the compiler to know which DOALL loops and arrays should be considered together. A connection graph is denoted by $CG(d, a, e)$ where d and a are two sets of vertices and e is a set of undirected edges. Each vertex in d represents one level of a DOALL loop in the program while each vertex in a represents an array. An edge in e connects a vertex in d to a vertex in a if any elements of the represented array are referenced in the represented

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

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


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

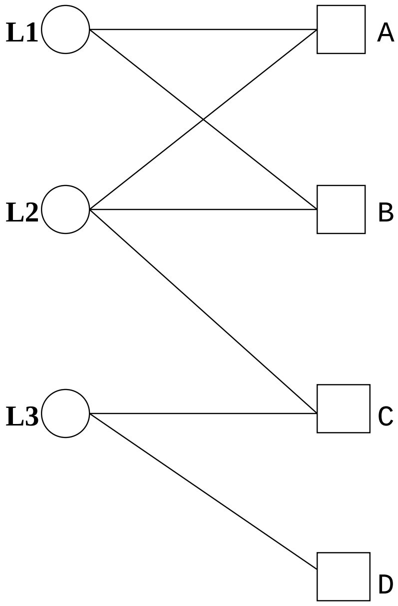


Figure 2: An example of a connection graph. The circles represent loop nodes while the boxes represent array nodes.

DOALL loop. Figure 2 shows a connection graph for a sample program segment. After constructing the connection graph, we find all connected components in this graph. A connected component of a graph is defined as a subgraph in which there is at least one path between any two nodes. Each connected component covers the arrays and loops that should be considered together when determining the orientations and displacements.

4.2 The Orientation Graph

For each connected component in CG, we first construct an undirected, weighted *orientation graph* OG, which is a new graph proposed by this work. The orientation graph augments the *component affinity graph* of J. Li and M. Chen [20] by adding nodes which represent DOALL loops to handle loop iteration assignment. We also introduce a "demand-driven" algorithm for the construction of the OG, which reduces the size of the graph, an important consideration when we analyze large programs.

Each node in OG represents an orientation choice, increasing or decreasing, for either an array A, or a DOALL loop L. Thus, each single-dimensional array A may generate up to two nodes, A^+ and A^-, in OG. If these two conflicting decisions both appear in OG, then we need to eliminate the one that is inferior. Likewise, each DOALL loop L may generate up to two nodes, L^+ and L^-, in OG. Only one of them will remain as the final decision. 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 may have up to $2n$ nodes representing the dimensions of the array and the possible orientation in each dimension.

Suppose an array A is referenced within a DOALL loop L. Their orientation choices, say A^+ and L^+, may result in a number of A references that are likely to be local, which we call 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. This cost represents the number of remote memory references that would arise if this particular preference of orientations is not granted. The larger the mismatch cost on an edge, the more important it is to grant the indicated preference. After constructing OG, we attempt to find a subgraph, SOG, which contains only one node for each DOALL loop or each array and whose edges have the maximum total mismatch cost. This problem is NP-hard and we use a heuristics for its solution. The heuristics itself is not the emphasis of this paper and the interested readers are referred to [24].

Obviously, if we include all possible orientation nodes for each array or each DOALL loop in the OG, then there exist a number of possible SOG's which are all equivalent. In order to reduce the size of the OG and the time spent on computing the SOG, our algorithm for constructing the OG inserts the nodes "on-demand". A new array orientation node is inserted in the OG only if it matches the orientation of an existing loop orientation node. Likewise, a loop orientation node is inserted only if it matches the orientation of an existing array orientation node. Whenever the algorithm can no longer make progress while there still exist DOALL loops whose orientations are undecided, the algorithm picks one of those loops and arbitrarily assigns it an orientation. (A different orientation would lead to equivalent, although different, SOG's.) Figure 3 shows the algorithm for building the OG. The

Input: A connected component, C, in the connection graph, the header information about the loops and the reference pattern of the arrays.

Output: An OG.
Method:
While there exists a DOALL loop, L, in C, which has not been represented in OG, do
Arbitrarily pick an orientation choice, say +, for L and
insert the node L^+ in OG and in a queue of DOALL loops, QL.
(Step 1) For each loop node, say L^+, in QL, remove it from QL.
Examine the patterns of references to array A in L.
If there are subscripts in the d-th dimension which increase with L's index variable and the node $A^{+,d}$ is not yet in OG, then insert the node $A^{+,d}$ both in OG (to match L^+) and in a queue of arrays, QA.
If there are subscripts in the d-th dimension which decrease with L's index variable and the node $A^{-,d}$ is not yet in OG, then insert the node $A^{-,d}$ both in OG (to match L^+) and in QA. (If array A has one dimension only, then we omit the notation of d in the orientation nodes.)

(Cont'd)

Figure 3: An algorithm for constructing the OG.

worst-case complexity of this algorithm is in the order of the product of the number of DOALL loops and the number of array references.

Figure 4 shows a simple OG example. Initially, the + orientation is arbitrarily picked for loop $L1$, which creates the node $L1^+$. The references to arrays A and B in $L1$ cause the insertion of A^+ and B^+, respectively. With A^+ in place, the references to $A[N-i]$ and $A[N-i+1]$ in loop $L2$ cause the insertion of $L2^-$ and an edge between $L2^-$ and A^+ with a mismatch cost of $2N$. On the other hand, the references to $B[i+2]$ and $B[i+4]$ in loop $L2$ cause the insertion of $L2^+$ and an edge between $L2^+$ and B^+ with a mismatch cost of $2N$. With $L2^+$ in place, the reference to $C[i,j]$ in $L2$ causes the insertion of $C^{+,1}$ and an edge between $L2^+$ and $C^{+,1}$ with a mismatch cost of $50N$. Similarly, an edge is drawn between $L2^-$ and a newly inserted $C^{-,1}$ which a mismatch cost of $50N$. The algorithm cannot make progress anymore at this point, although the orientation of $L3$ is still

(Cont'd)

Draw a new edge between L^+ and each of its matching array orientation nodes, no matter whether the matching node is newly created.
Annotate the new edge by the estimated mismatch cost (discussed later).

(Step 2) For each array node, say $A^{+,d}$, in QA, remove it from QA.
For each DOALL loop, say LL, which contains references to array A such that LL's index variable appears in the d-th dimension, examine the coefficients of the index variable. If there are coefficients that are positive and the node LL^+ is not yet in OG, then insert the node LL^+ in OG and in a queue of loops, QL.
If there are coefficients that are negative and the node LL^- is not yet in OG, then insert the node LL^- in both OG and QL.

Draw a new edge between $A^{+,d}$ and each of its matching loop orientation nodes, no matter whether the matching node is newly created.
Annotate the new edge by the estimated mismatch cost (discussed later).

If QL is nonempty, repeat the above from Step 1.

End of while loop.

Figure 3: An algorithm for constructing the OG.

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

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


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

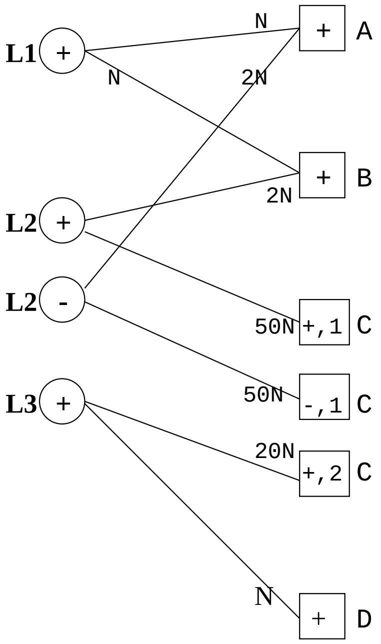


Figure 4: An example of an orientation graph.

undecided. We arbitrarily pick + and insert $L3^+$, which causes the insertion of $C^{+,2}$ and D^+. Figure 5 shows the SOG, the result of reducing the OG in Figure 4 to maximize the total mismatch cost. The orientation decisions indicated by the SOG are A^-, B^+, $C^{+,1}$, D^+, $L1^+$, $L2^+$ and $L3^+$.

Since the SGI parallel run-time library used in our simulation experiments (*c.f.* Section 6) can handle only singly-nested DOALL loops, our compiler must sequentialize all but one level of a DOALL loop. If several levels of a multiply-nested loop can be parallelized, we select one level to keep as a DOALL loop when we determine the DOALL loop orientations. The other levels are changed back to DO loops. We handle multiply-nested DOALL loops similarly to the way we handle multi-dimensional arrays in the last section. We create a plus and minus node for each level of a multiply-nested DOALL loop. Figure 6 shows a partial OG for a segment of codes. Only one orientation choice at one loop level for each multiply-nested DOALL loop is included in the final SOG.

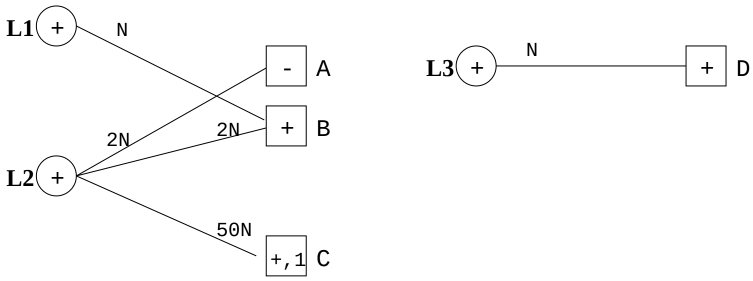


Figure 5: The resulting subgraph SOG for the OG in Figure 4.

```
L11   DOALL i = 1, N
L12     DOALL j  = 1, N
          A(i, j) = A(i, j)+B(i, j)
        ENDDO
      ENDDO

L21   DOALL j  = 1, N
L22      DOALL i = 1, N
           A(i, j) = A(i, j)+B(i, j)
         ENDDO
      ENDDO
```

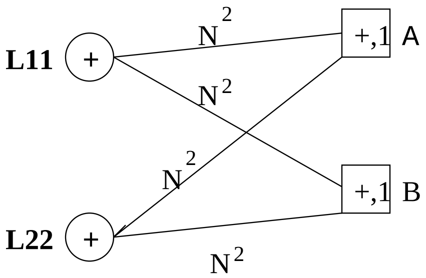


Figure 6: An example of multiply-nested DOALL loops and how they are represented in an OG.

4.3 The Displacement Graph

The displacements used in the assignment of array data and DOALL loop iterations to the VP's are determined in a way similar to the one for orientation, using a *displacement graph*, DG, in place of the orientation graph. This displacement graph is also unique to our work. For each orientation node in the final SOG we computed previously, we create a number of displacement nodes, representing different potential displacement decisions. Those orientation nodes rejected from the orientation graph no longer need to be considered. Since the algorithm for constructing the DG is analogous to that for constructing the OG, we omit its details. Instead, we use the sample program segment in Figure 4 to illustrate how the algorithm works.

The displacement graph for our example is shown in Figure 7. Initially, we arbitrarily assign a displacement of zero to $L1^+$, which causes the insertion of a node for B^+ with a displacement of -1 such that the reference to $B[i+1]$ in loop $L1$ will find the data in the same VP which executes the i iteration of $L1$. Note that the reference to $A[i]$ in the same loop is ignored because its favored A^+ orientation was rejected earlier. With the displacement of -1 for B^+ in place, the references to $B[i+2]$ and $B[i+4]$ in loop $L2$ cause the insertion of two nodes for $L2^+$ with displacements of $+1$ and $+3$ respectively. These two new loop displacement nodes in turn induce a number of array displacement nodes for A^-, namely 0, $+1$, $+2$ and $+3$. The same loop displacement nodes also induce two displacement nodes for array orientation $C^{+,1}$, namely $+1$ and $+3$. At this point, the algorithm cannot make progress any further, although the displacement for $L3^+$ is yet to be decided. We arbitrarily pick a displacement of zero for $L3^+$, which leads to the displacement of $+1$ for D^+. Each edge in the DG matches a loop displacement decision to an array displacement decision, annotated by an estimated number of remote references which will result if these two nodes are not matched.

The worst-case time complexity for constructing the DG is in the order of the product of the number of DOALL loops, the number of array references and the maximum number of different displacement patterns in the subscripts of each array. Again, we attempt to find a subgraph, SDG, such that it contains only one node for each DOALL loop or each array and that the total mismatch cost is maximum. The shaded nodes in Figure 7 show

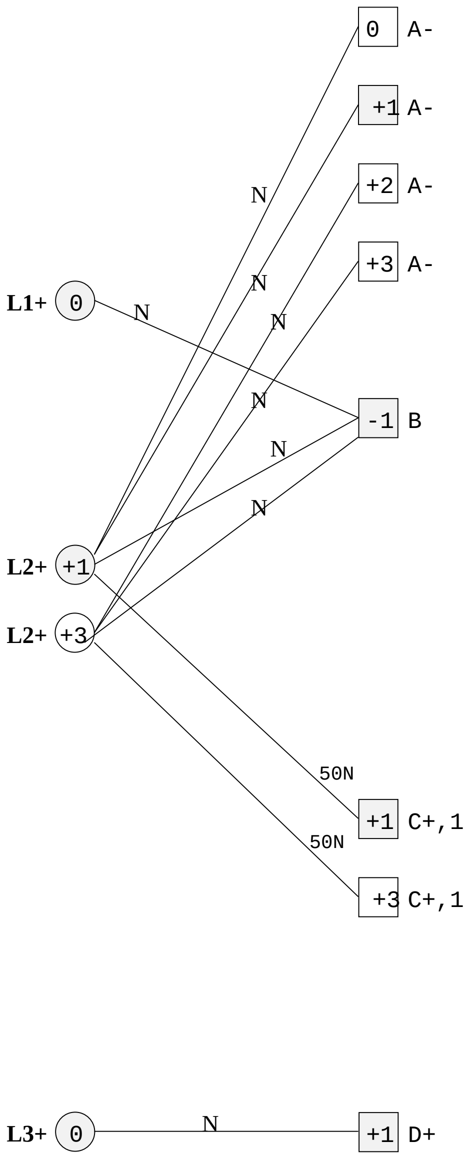


Figure 7: An example of a displacement graph. The shaded nodes are the ones that will be in the resulting subgraph SDG.

```
propagate_iters (v, num_iters)

v → total_iters = num_iters
if v is a COMPOUND DO
  propagate_iters (v → child, v →total_iters * number iterations in v)
end if
for each successor, s, of v do
  propagate_iters (s, v → total_iters)
end for
```

Figure 8: An algorithm for propagating the information on loop iterations through an HSG.

a final SDG. We use a linear-time heuristic algorithm to find a SDG [24], but we leave out the algorithm because it is not the focus of this paper.

4.4 Estimating the mismatch costs

The mismatch cost associated with each edge is the main factor in deciding what nodes are included in SOG and SDG. Estimating the mismatch cost is based on the number of times each array reference is accessed, which is estimated by accumulating the number of loop iterations that may be executed and propagating that number through the HSG. The sum of loop iterations at each basic block represents the number of accesses to each array reference that occurs in the basic block. Using the HSG structure in Panorama, the propagation can be easily performed using the procedure in Figure 8. We start the propagating step by passing it the entry node of the HSG for v and one for num_iters. When we see a compound DO node, we calculate the total number of iterations accumulated multiplied by the number of iterations of the current loop and propagate the result to the DO loop's control flow subgraph. The notation $v \rightarrow child$ in Figure 8 represents the entry node of v's control flow subgraph. The accumulated number of iterations are also propagated to all successors of v. The probability of each IF statement branch being taken is averaged from profiling information. The product of this probability and the accumulated iteration number is considered as the frequency for the basic block to be

Table 1: The size of the connection graphs constructed and the number of connected components found. Note that data for `mgrid` is incomplete.

Program	*Size in lines (excluding comments)*	*# of CG nodes*	*# of CG edges*	*# of disconnected subgraph*
adi	150	19	35	2
swm256	288	29	76	1
tomcatv	126	15	24	3
mgrid*	505	11	1	1
su2cor	1783	124	152	20
hydro2d	4222	106	47	67
dyfesm	4684	51	18	33
flo52q	2627	77	45	45
qcd	2635	62	54	10
track	2171	89	74	23

executed. This frequency may contain symbolic terms. Symbolic arithmetic expressions are evaluated by using one of the symbolic evaluation utility routines in Panorama (*c.f.* Section 3).

4.5 Complexity

There are several parts that contribute to the complexity of the process of assigning the DOALL loop iterations and the array data to the VP's. The calculation of the mismatch costs requires traversing the HSG. By traversing in a bottom-up order, each node in the HSG only needs to be visited once to obtain enough information for estimating the mismatch costs for both orientation and displacement graphs. The main part of the assignment process consists of creating OG nd DG and finding the SOG and the SDG. The computation is dominated by the construction of DG whose time complexity was discussed earlier. (Our heuristic algorithm for finding SOG and SDG is a linear algorithm with respect to the number of nodes and edges [24].) Table 1 shows the size of the connection graph and the number of connected components found in the connection graph for several numerical programs. We will give more descriptions of some of these programs in Section 6. This table also gives the program size in term of the number of lines excluding comments. For each connected component, we build one orientation graph and one displacement graph. Table 2 shows the largest orientation and displacement graphs in term of the number of

Table 2: The size of the orientation and displacement graphs constructed. (Data for `mgrid` is incomplete.)

	# of nodes		# of edges	
Program	*Largest OG*	*Largest DG*	*Largest OG*	*Largest DG*
adi	48	174	64	308
swm256	86	329	152	759
tomcatv	40	147	48	198
mgrid*	6	22	2	11
su2cor	194	858	250	1375
hydro2d	26	99	24	132
dyfesm	10	44	6	33
flo52q	30	123	30	154
qcd	64	325	60	319
track	48	242	52	286

nodes and edges, that are built for each program. Note that `mgrid` contains a case of parameter mapping (*c.f.* Section 5) that was not implemented. Therefore, the size of the graphs for `mgrid` is smaller than they should be. The practical measure of an algorithm's complexity is its running time. With the assignment algorithm for DOALL loop iterations and array data enabled in the Panorama compiler, the numerical programs in the Table 1 require no more than one to two seconds of additional compile time on an SGI Indy workstation.

5 Parameter Mapping

Due to the interprocedural nature of the algorithm, parameter mapping is important for associating formal parameters to the actual data and for summarizing the mismatch cost of an actual parameter at a call site. We discuss the parameter mapping issue below.

In a Fortran program, there are two classes of variables. One class of variables are allocated to memory locations in the program's virtual address space. For convenience, we call this class *actual* variables. The other class of variables are formal parameters in subroutines and they are not allocated to memory locations. In the example in Figure 9, the variables `X`, `Y`, `C`, and `D` are actual variables. Variables `A` and `B` on the other hand are formal

```
Subroutine Foo (A, B)
Dimension A(1), B(1)
...
DOALL i = ...
  ...  = A(i) + B(i+1)
ENDDO

Program Example
Dimension X(100), Y(101), C(100), D(101) ...
CALL Foo (X, Y)
...
CALL Foo (C, D)
...
```

Figure 9: Example of parameter mapping. Formal parameters A and B must be mapped to actual data.

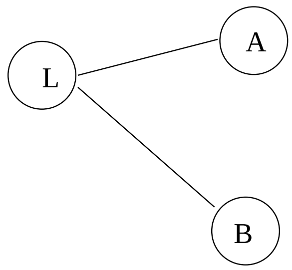
parameters. The addresses of `A` and `B` depend on the actual arguments that are passed at the call to subroutine `Foo`. In this example, `A` and `B` map to `X` and `Y` respectively at the first call statement and to `C` and `D` respectively at the second call statement. Hence, the memory location represented by a formal parameter may vary from one invocation of the subroutine to another. Subroutine in-lining [7, 8, 11] can eliminate all formal parameters, and subroutine cloning [6] can result in a unique virtual address for every formal parameter. However, in-lining and cloning may make the resulting object code overly large and, thus, are performed to limited extent in practice.

When we determine the orientations and the displacements for array data, the data being distributed across the VP's are the elements of actual variables that would map to virtual addresses. In Figure 9, the elements of array variables `X`, `Y`, `C`, and `D` are distributed among the processors. For formal parameters such as `A` and `B`, the algorithm needs to know which actual variables they map to. Note that a formal parameter may map to another formal parameter. However, if we follow through the chain of calls, eventually it maps to one or more actual variables. For example, in Figure 10, before creating an array node for `AA`, we need to know if `AA`

```
Dimension A(N,M), B(N,M)
...
CALL Foo (A)
...
CALL Foo (B)
...

Subroutine Foo (AA)
Dimension AA(N,M)

DOALL i = 1,M
  DO j = 1,N
    ...  = AA(i, j)
  ENDDO
ENDDO
```



(b) Connection graph for the example code

Figure 10: One mapping that is commonly seen in numerical programs.

actually maps to other actual variable(s). In this example, `AA` maps to `A` and `B` so we do not need to create a node for `AA` itself. Instead, we connect an edge between array node A to the loop node L and an edge between B and L to indicate that both actual arrays `A` and `B` are referenced in the DOALL loop.

The example in Figure 10 represents one of the most common types of parameter mapping we found. The formal parameter has the same number of dimensions as the actual parameter. The size of each dimension of the formal parameter is either the same or smaller than that of the actual parameter. In other words, the formal array represents a sub-region of the actual array. In this case, we simply map the formal array name to the actual array name and add certain offsets to the subscripts of the formal array so that references to the formal array elements correspond to the correct actual array elements. In our example, since the size of `AA` is the same as that of `A` and `B`, no adjustment in the subscripts is necessary.

Another common type of parameter mapping is shown in Figure 11. Here, the number of dimensions of the formal parameter is less than that of the actual parameter. In the example, the formal array `AA` is a one-

```
Dimension A(N,M)
...
DO i = 1, N
  CALL Foo (A(i, 1))
ENDDO
...

Subroutine Foo (AA)
Dimension AA(M)

DOALL i = ...
  ...  = AA(i)
ENDDO
```

Figure 11: Another common case of parameter mapping. The number of dimensions of the formal parameter is less than that of the actual parameter.

dimensional array while the actual array `A` is a two-dimensional array. At the call site, only one dimension, namely the columns of `A`, is passed in to the calling subroutine. Hence, `AA` refers to the second dimension of A and needs to be mapped as such. When the orientation (or displacement) graph is constructed, this mapping information is used to label the edges between the proper nodes representing a loop orientation choice and an array orientation choice with the correct mismatch costs. For this example, there will be edges with positive mismatch costs connecting the orientation node $A^{+,2}$ and $A^{-,2}$ with the loop node. With the given information in Figure 11, the second dimension of array `A` should be distributed among the processors.

An example of a third common type of parameter mapping found in numerical programs is shown in Figure 12. The number of dimensions of the formal parameter is greater than that of the actual parameter. This type of parameter mapping is more difficult to handle. If each formal array can be mapped into unique regions of the actual arrays, the actual array can be divided and then renamed. In the example in Figure 12, `A` can be divided into two arrays, the first with 100 elements and the last with 400 elements. The two arrays can then be renamed A_{AA} and A_{BB} respectively to refer

```
Dimension A(500)
...
CALL Foo (A(1))
CALL Foo2 (A(100))
...

Subroutine Foo (AA)
Dimension AA(10,10)
...

Subroutine Foo2 (BB)
Dimension BB(4,100)
...
```

Figure 12: A case of parameter mapping in which the number of dimensions of the formal parameter is greater than that of the actual parameter.

to formal array AA and BB. The new arrays take on the same dimension information as the formal arrays.

However, if a formal array refers to a region in the actual array that overlaps with a region referred to by another formal array, we must choose of which region of the actual array should be renamed to a new array. This decision can be made during phase I of the alignment algorithm. At first, we create a new array name for each region, whether the regions overlapped or not. When constructing the orientation graph, nodes representing the orientation choices of the new overlapping arrays must be considered as conflicting nodes, similar to the way how multi-dimensional arrays are handled. In the final SOG, the remaining nodes represent the orientation choices as well as how the original array should be divided.

The quality of parameter mapping can significantly affect how successful the alignment algorithm will be. Table 3 shows the number of array references encountered and the percentage that needs parameter mapping for a list of numerical programs. In the current implementation of the alignment algorithm, the first two cases are handled. The last case which requires dividing an array is not yet implemented. If such a case is encountered (as

Table 3: The percentage of parameter references in the test programs.

Program	*# of array refs*	*% parameter refs*
adi	75	28
swm256	160	0
tomcatv	67	0
mgrid	130	86
su2cor	2095	16
hydro2d	919	28
dyfesm	338	54
flo52q	550	49
qcd	118	40
track	131	16

in `mgrid` and some programs in the Perfect benchmarking suite [5]), our current implementation ignores these array references.

6 Simulation Results and Summary

Through execution-driven simulations, we examine the effect of static scheduling of DOALL loops and data allocation on a CC-NUMA architecture. These results are intended as an overview to show that, as expected, alignment of DOALL loops and array data can improve the performance considerably. More detailed discussions can be found in [24].

The Panorama parallelizing compiler (*c.f.* Section 3) is used to parallelize and instrument the benchmarking programs for the experiments. It also provides the framework for static scheduling and data allocation. After a program is parallelized and instrumented by Panorama, it is compiled by SGI's `f77` compiler to a parallel object code executable on the SGI Challenge multiprocessor. The SGI Challenge provides run-time libraries which implement the loop scheduling methods mentioned in the introduction. The executable object code is simulated by *NUMASim*, an execution-driven CC-NUMA cache simulator. NUMAsim consists of two main parts: the MINT event generator front-end [29] and our detailed CC-NUMA cache simulator back-end.

The MINT front-end interprets each instruction in the executable object code and schedules events such as loads and stores for the back-end to simulate. MINT adopts a simple processor model in which each non-load/store instruction takes one cycle and the processor stalls at a cache miss. An event originating from the instructions may trigger additional events internal to the memory hierarchy. Such events are also scheduled by MINT upon the request from the back-end. For example, while processing a load event, a cache miss can cause another event to handle reading from local or remote memory. When all events triggered by the load or the store are processed, the back-end sends a control signal back to MINT so it can continue to interpret the next instruction in the executable object code. This process continues until the interpreted code runs to stop.

Since Panorama is a source-to-source compiler, it does not generate virtual addresses for the machine code. Instead, Panorama inserts data allocation directives in the program, and in the NUMAsim simulator, we re-map the virtual addresses, generated by the SGI's *f77* compiler, to reflect the selected data allocation scheme [24]. As mentioned previously, the directives inserted by Panorama allow the simulator to identify the starting address and the size of each array and what data allocation scheme to be used for that array. Using this information, the simulator changes the virtual addresses of array data on the fly. In addition, a *page-coloring* page mapping scheme [17] is adopted for the mapping from a virtual page to a physical page. Because the simulator re-maps virtual addresses of global array data (which occurs in all of our test programs) before the instruction is interpreted, we can verify the correctness of the mapping processes by comparing the program's output before and after re-mapping. Most errors in the mapping process will cause a core dump or yield different output compared to an actual run on an SGI workstation.

All experiments performed in this paper use the following default settings for the NUMA cache simulator. The simulated system contains 16 processors. Each processor has a 1M byte, 4-way set-associative external cache. Each cache line consists of 32 bytes or 8 words. A bandwidth of one word per processor cycle is assumed for each memory module and for the network. Table 4 shows the latency for different types of events that are associated with cache and memory accesses. These numbers are chosen based on several known CC-NUMA designs. A load or a store may trigger one or more of these events. For example, a cache-read hit normally requires one access to the cache, resulting in a delay of one cycle. On a cache-read

Table 4: Latency for various delays, where *lsize* is the line size in words (which is 8 for our studies).

Event type	*Latency in cycles*
Cache access	2
Memory access	$7 + lsize$ - 1
Directory access	1
1-way network delay	20

miss, if the requested memory block is not dirty, it is fetched from the memory and then sent back to the cache.

The actual delay for cache and memory accesses may vary depending on the availability of the cache ports and the memory ports. Our simulator assumes a single port for each cache and each memory module. This means that while the cache (or the memory module) is satisfying a request, no other requests to this cache can be processed. Those outstanding requests are queued and processed in the order that they are received.

6.1 The test programs

Several numerical Fortran programs are used for the simulations, most of which are from the SPEC benchmarks. These programs are chosen for their relatively high degree of parallelism and their reasonable simulation time. This section provides a short descriptions of these test programs.

`ADI` is a numerical algorithm used to solve partial differential equations [26]. The main data in `ADI` is nine 256x256 arrays of double words. There are two main DOALL loop nests with 254 iterations at each level. One array is swept along the columns in one loop nest and then along the rows in the other. No matter how this array is allocated to processors, some memory misses will occur. For the other arrays, the first loop nest sweeps across the columns in half of the arrays while the second loop nest sweeps along the rows in the rest. For these arrays, whether rows or columns are distributed across the processors depends on which loop nest accesses the arrays.

`Swm256` (shallow water model) is a weather prediction program taken from the SPEC92 benchmark suite. The main data consists of 14 arrays of 257 x 257 words. The number of the iterative time steps were reduced from

```
DOALL J=2,N-1
  DO I=2,N-1
    XX=X(I+1,J)-X(I-1,J)
    YX=Y(I+1,J)-Y(I-1,J)
    XY=X(I,J+1)-X(I,J-1)
    YY=Y(I,J+1)-Y(I,J-1)
    ...
  ENDDO
ENDDO
```

Figure 13: An example of the referencing pattern occurred in `tomcatv`.

1200 to 60 to shorten the simulation time. Most DOALL loop nests in this program are two-level deep, with each level containing 256 iterations. For the experiments, the outermost level in each loop nest is kept as a DOALL loop. Since the outer loop index sweeps across the columns of the arrays, the second dimension of each array should be distributed across the processors.

`Tomcatv` is a mesh generation solver also taken from the SPEC92 benchmark suite. The main data consists of seven arrays of 257 x 257 double words. The main DOALL loop nests are two-level deep, with each level containing 256 iterations. The outer loop index sweeps across the columns in six of the arrays and across the rows in one other array. The referencing pattern in the main loop nests is slightly different from `swm256`. Here, one neighboring column from each direction and one neighboring row from each direction are accessed as well as the current column or row. Figure 13 shows an example of this referencing pattern. Again, the time steps are reduced from 100 to 5 to speed up simulation runs.

`Hydro2d` is another SPEC92 benchmark program from the area of astrophysics. In this program, hydro-dynamical Navier Stokes equations are solved for the computation of galactical jets. The most time consuming subroutines are FILTER and FCT which consume about 46% and 16% of sequential execution time. Both of them consist of several two-level loop nests and are called 3200 times with the number of time steps set to two. More than 95% of the loop nests in the program can be parallelized, among which more than 90% are visited by column.

`Mgrid` is a 3-dimensional potential field computation program from the SPEC95 suite. The main data consists of three arrays, one of which has

287496 double words and the rest of them have 333944 double words. Even though the data are declared as a single dimension, they are actually passed to subroutines as three dimensional array parameters. To obtain a reasonable simulation time, the time steps are reduced from 40 to two. We eliminate the outermost loop which does nothing but repeatedly exercising the machine with the same input producing the same result.

6.2 Data collected

We have collected data for the five programs listed in Section 6.1. Most programs use all 16 processors with the exception of `hydro2d` which uses only 8 processors because of its relatively fine granularity. The program speedup of parallel machine code, under a particular simulation setting, is the execution time of the optimized code of the original sequential program over the parallel execution time. We use the `f77 -O2` command to produce the optimized sequential codes. The `-O2` flag is used for producing the parallel code as well, in conjunction with other flags specific to parallel code generation. We interchange DOALL loops with the outer sequential loops whenever possible, to reduce scheduling overhead. Loops are not interchanged in those sequential codes, since interchange did not improve the sequential execution.

The results achieved by our compiler have been compared favorably against several straightforward data allocation schemes which do not perform alignment [24]. To simplify the graphs which exhibit the performance curves, however, we pick only two schemes as reference points in this paper. The first reference scheme used for comparison is *flat-space interleave*, which partitions the entire address space into chunks of cache line size. These chunks are then distributed to the processors by interleaving. The second reference scheme is *per-array block*, which evenly divides the linear address space of each array into p chunks before assigning them to the p processors. The size of the array is always rounded up to a multiple of the cache line size times p. The main purpose of using these reference schemes is to indicate how the program performance may degrade if the compiler ignores the issue of aligning DOALL loop iterations with data.

Notice that any alignment arrangement can easily be destroyed by dynamic loop scheduling such as *gss* and *dynamic* (*c. f.* Section 1). The results in this section also show how much the program performance may suffer due to dynamic loop scheduling.

The simulation results are shown in Figures 14 through 18, where the curve labeled by "data-task co-allocation" represents our assignment scheme for DOALL loop iterations and array data. The "interleave" marking on the x-axis represents chunk interleave scheduling with the iteration chunk size determined at compile time to agree with the data allocation schemes. The "dyn4" marking represents self-scheduling with an iteration chunk size of four. The "block" marking represents SGI's *simple* scheduling method (*c.f.* Section 1.)

Each sub-figure shows one of three performance metrics: the speedup over the best sequential execution time, the cache miss ratio, which is the percentage of cache misses out of all loads and stores, and the memory miss ratio, which is the percentage of cache misses which result in remote memory references out of all cache misses. (Note that memory miss ratio has a stronger effect on the execution speed when the cache miss ratio is higher.) As a reference point, we show a curve (marked as "no network delay") for the speedup by setting the network latency to zero, assuming the simplest flat-space interleave allocation scheme. Our data-task alignment algorithm always distributes data across the right array dimensions for the DOALL loop nests found in the shown benchmarks and, as the result, it consistently yields relatively low cache miss ratios and memory miss ratios, and it delivers high speedup when compared with the reference schemes.

For `swm256`, interleaving the data by the cache line size across the processors achieves relatively poor program speedup, as the "flat space" curve shows. Even when no network delay is charged, the speedup is still lower than those achieved by *per-array block* allocation and by our compiler scheme. This is mainly due to the relatively high cache miss rate when the data are interleaved for this program, even though the interleaving is performed using a unit size of a cache line in order to reduce cache false-sharing. Both our compiler scheme and the *per-array block* scheme have cache miss ratios lower than 0.5%. Our scheme has a much lower memory miss ratio than *per-array block* and the speedup of our scheme is higher.

For program `tomcatv`, our scheme performs worse than the *per-array block* schemes when the loops are scheduled by using *block* scheduling. Based on the inspection of the code and our generated data mapping, we believe that our scheme does not align the cache line boundaries as well as the *per-array block* scheme in this case. The selection of the optimal displacement is slightly more difficult than selection of orientation, due to the subtle effect of aligning the cache line boundaries. We will study more details in the future work.

For program `adi`, *per-array block* performs significantly poorer than our compiler scheme and it does not even outperform *flat space* in this case. For programs `mgrid` and `hydro2d`, *per-array block* and our compiler scheme allocate data identically or almost identically, and they give identical or almost identical results. Overall, we note that our scheme performs consistently well for all programs, while none of the reference schemes do.

Even when the array dimension being distributed matches the DOALL loop level, the misalignment between the displacements of data allocation and DOALL loop iterations can make the performance suffer greatly. For example in `swm256` (*c.f.* Figure 14), dynamic scheduling causes data misalignment for the good allocation schemes, resulting in a sharp drop in the program speedup.

6.3 Summary

We have presented a compiler scheme for loop scheduling and data allocation. Taking advantage of interprocedural propagation of array reference information, we build two decision graphs, namely the orientation graph and the displacement graph, to represent globally and interprocedurally the remote reference penalties associated with different choices for loop scheduling and data allocation. the compiler generates data allocation directives and schedules DOALL loops. We then obtain a solution which, according to these graphs, minimizes remote memory references on CC-NUMA multiprocessors. The experiments which we performed in the Panorama parallelizing compiler indicate that the proposed compiler scheme is efficient. Moreover, simulation results suggest that this scheme can achieve quite good parallel program performance. For future work, we plan to perform experiments on commercial CC-NUMA machines.

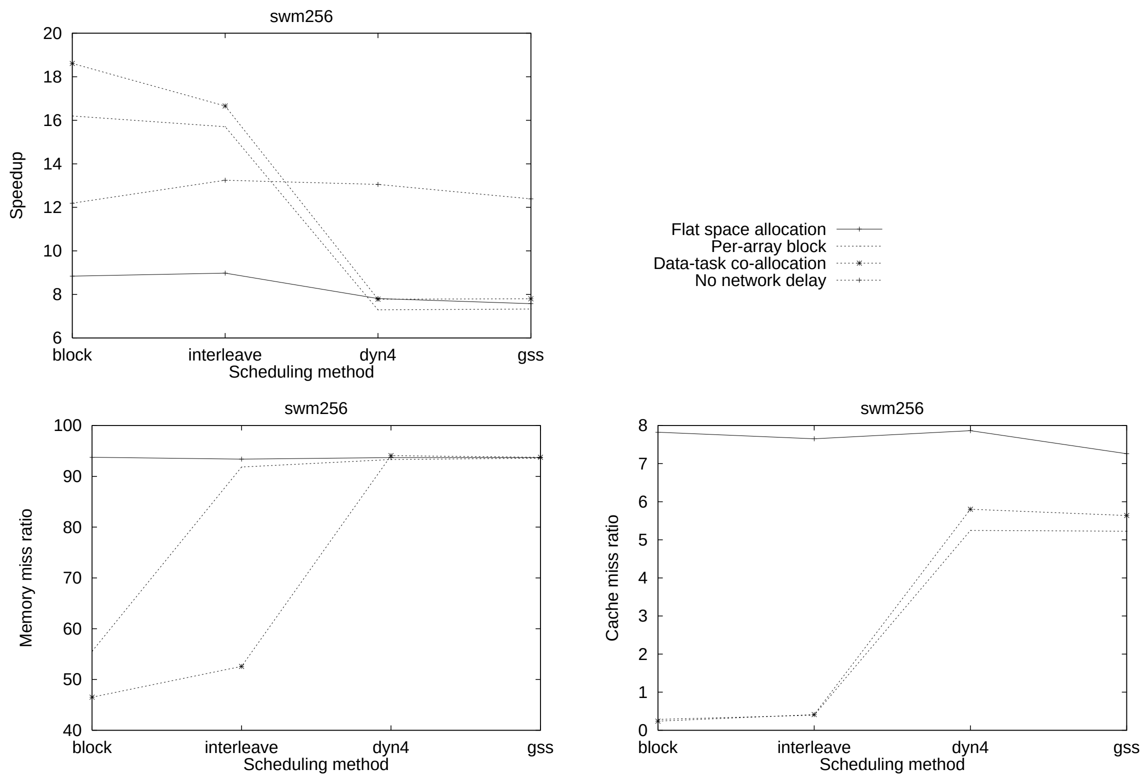


Figure 14: Performance comparison for `swm256`.

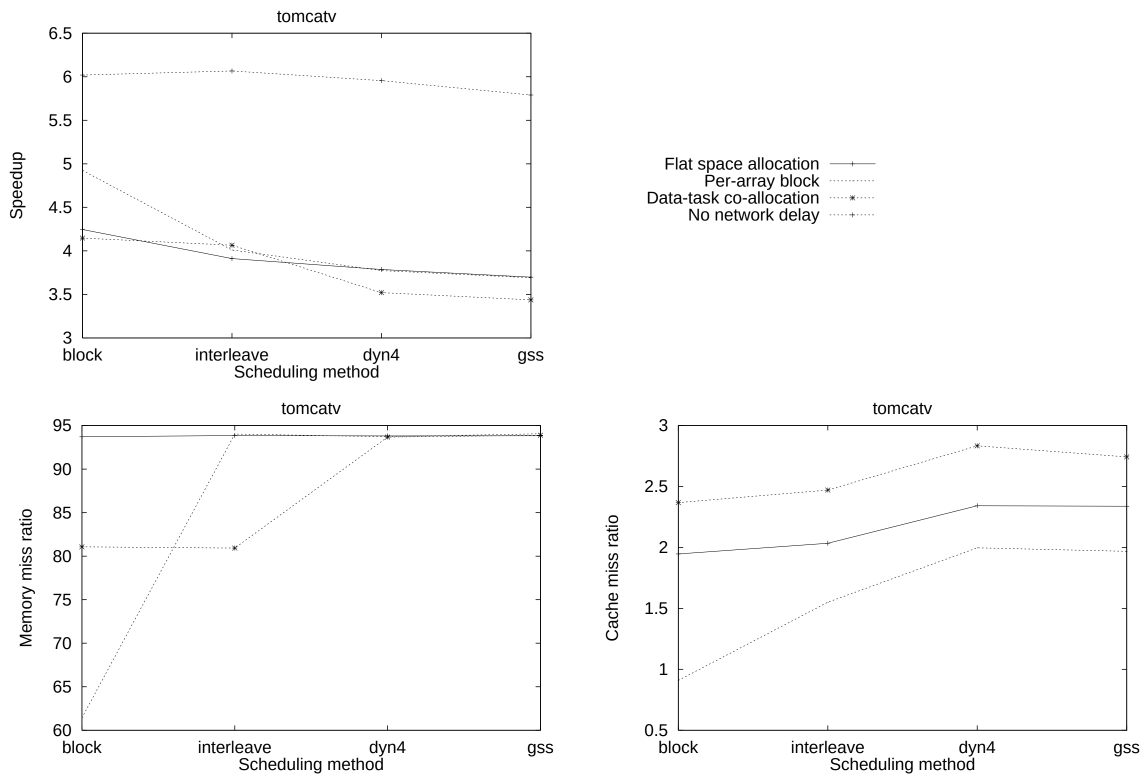


Figure 15: Performance comparison for `tomcatv`.

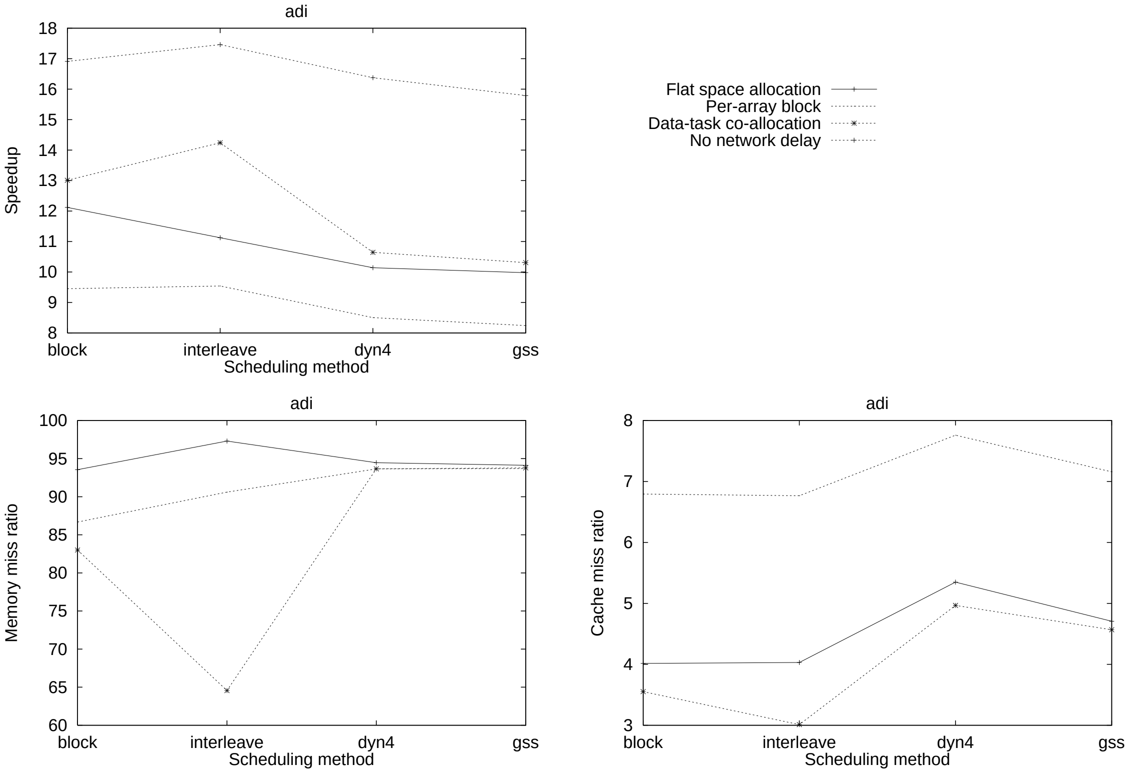


Figure 16: Performance comparison for `adi`.

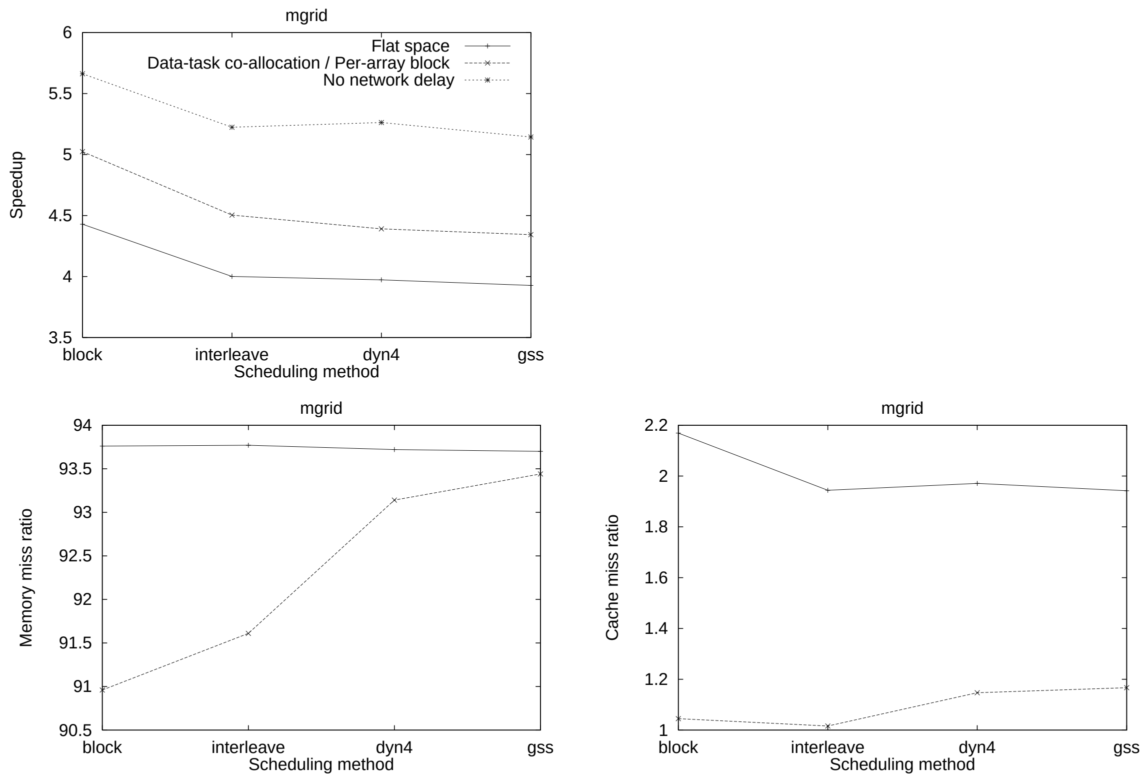


Figure 17: Performance comparison for `mgrid`.

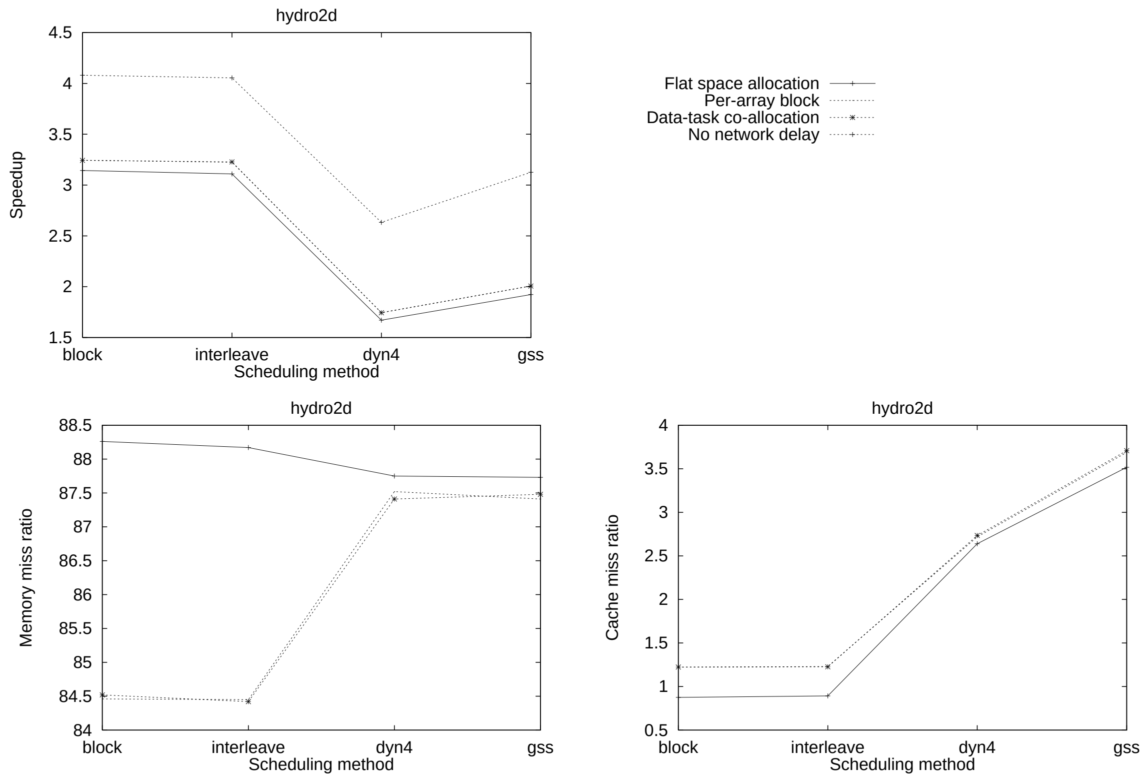


Figure 18: Performance comparison for `hydro2d`.

Acknowledgement

This work is supported in part by the National Science Foundation, Grants NSF/CCR-9210913 and NSF/CCR-950254.

References

[1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors. In *Proc. International Conference on Parallel Processing*, volume I: Architecture, pages 2–11, St. Charles, IL, 1993.

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. An overview of a compiler for scalable parallel machines. In *Proc. of the Sixth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1993.

[3] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proc. of Fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 166–178, July 1995.

[4] 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.

[5] M. Berry, D. Chen, P. Koss, D. Kuck, and S. Lo. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD-827, University of Illinois, Urbana, IL, May 1989.

[6] K. Cooper, M. Hall, and K. Kennedy. Procedure cloning. In *Proc. IEEE International Conference on Computer Language*, Oakland, CA, April 1992.

[7] K. Cooper, M. Hall, and L. Torczon. An experiment with inline substitution. *Software–Practice and Experience*, 21(6):581–601, June 1991.

[8] K. Cooper, M. Hall, and L. Torczon. Unexpected side effects of inline substitution - a case study. *Letters on Programming Languages and Systems*, 1(1), March 1992.

[9] G. W. Elsesser, V. N. Ngo, S. Bhattacharya, and W.-T. Tsai. Processor preallocation and load balancing of doall loops. *Journal of Supercomputing*, 8:135–161, 1994.

[10] J. Gu, Z. Li, and G. Lee. Experience with efficient array data-flow analysis for array privatization. In *Proc. 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–167, June 1997.

[11] A. Holler. *A Study of the Effects of Subprogram Inlining.* PhD thesis, University of Virginia, Charlottesville, VA, May 1991.

[12] C.H. Huang and P. Sadayappan. Communication-free hyperplane positioning of nested loops. In *Languages and Compilers for Parallel Computing*, pages 186–200. Springer-Verlag, 1992.

[13] Silicon Graphics Inc. *POWER Fortran Accelerator User's Guide.*

[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] Y. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Languages and Compilers for Parallel Computing*, pages 344–358. Springer-Verlag, 1992.

[16] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proc. Supercomputing '95*, December 1995.

[17] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indixed caches. *ACM Trans. on Computer Systems*, 10(4), November 1992.

[18] D. Kulkarni, K.G. Kuman, A. Basu, and A. Paulraj. Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *ACM Int. Conf. on Supercomputing*, pages 206–215, June 1991.

[19] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers '90: The Third Sym. on the Frontiers of Massively Parallel Computation*, pages 424–432. IEEE, October 1990.

[20] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *J. Par. and Dist. Computing*, 13:213–221, 1991.

[21] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA computers. *ACM Trans. on Computer Systems*, 11(4), November 1993.

[22] Z. Li and T. N. Nguyen. An empirical study of the work load distribution under static scheduling. In *Proc. International Conference on Parallel Processing*, volume II: Software, St. Charles, IL, 1994.

[23] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proc. of the 24th SIGPLAN-SIGACT symposium on Principles of Programming Languages*, January 1997.

[24] T. N. Nguyen. Interprocedural compiler analysis for reducing memory latency. Technical report, PhD thesis, University of Minnesota, November 1996.

[25] C. Polychronopoulos and D. J. Kuck. Guided self scheduling: A practical scheduling scheme for parallel computers. *IEEE Trans. on Computers*, C-36(12):1425–1439, December 1987.

[26] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing (Fortran Version).* Cambridge University Press, 1989.

[27] V. Sarkar and G. R. Gao. Optimization of array accesses by collective loop transformations. In *ACM Int. Conf.s on Supercomputing*, pages 194–204, June 1991.

[28] P.-S. Tseng. A parallelizing compiler for distributed memory parallel computers. Technical Report CMU-CS-91-121, PhD thesis, Carnegie Mellon University, May 1989.

[29] J. E. Veenstra and R. J. Fowler. MINT tutorial and user manual. Technical Report 452, University of Rochester, June 1993. Dep. of Computer Science.